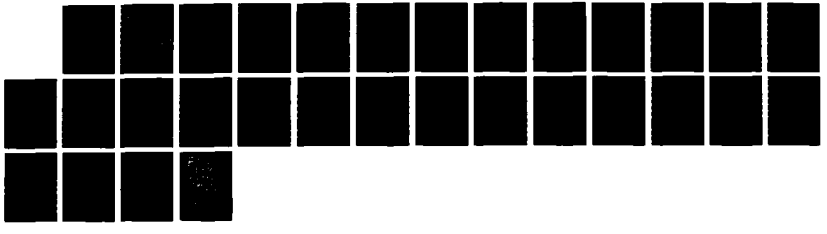
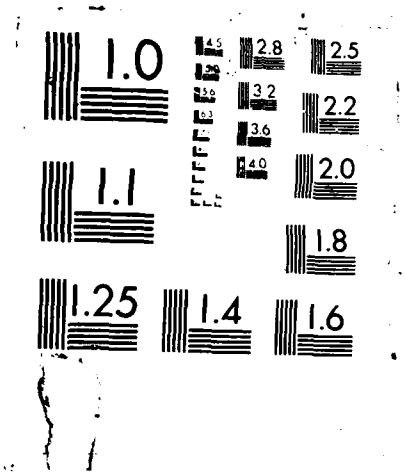
AD-A189 202 DESIGN OF THE CONSUL PROGRAMMING LANGUAGE(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE D BALDWIN FEB 87 TR-208 DACA76-85-C-0001 1/1

UNCLASSIFIED F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(4

DTIC FILE COPY

AD-A189 202

Design of the CONSUL Programming Language

Doug Baldwin
Cesar Augusto Quiroz Gonzalez
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 208
February 1987

Department of Computer Science
University of Rochester
Rochester, New York 14627

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 208	2. GOVT ACCESSION NO. A189 202	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Design of the CONSUL Programming Language		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Doug Baldwin		8. CONTRACT OR GRANT NUMBER(s) DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Rochester Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Project Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE February 1987 13. NUMBER OF PAGES 24
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT *(of this Report)* Distribution of this document is unlimited.		Accession For NTIS GRA&I ☑ DTIC TAB ☐ Unannounced ☐ Justification By Distribution/ Availability Dist A-1
17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*		
18. SUPPLEMENTARY NOTES None		DTIC COPY INSPECTED
19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)* parallel computation, constraint-based programming, logic programming, constraint languages		
20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)* We study the problem of automatically exploiting parallelism in computer programs, with particular emphasis on linguistic barriers to parallelism detection. Although functional languages and Prolog have many desirable characteristics in this respect, we find that they are not entirely ideal. We therefore offer constraint-based programming as a generalization of logic programming. By virtue of the more flexible ways in which they allow relations to be defined, constraint languages support more natural descriptions of potentially parallel algorithms than do existing logic or functional		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT (Continued)

languages. We introduce a prototype constraint language called CONSUL, which demonstrates features that we feel make constraint languages well-suited for general-purpose programming of multi-processors. The extra expressiveness of constraint languages comes at a price, namely that satisfaction of general constraints can be much more difficult than satisfaction of predicates for a language like Prolog. Nonetheless, we believe that effective compilers for constraint languages can be built, and we outline some ideas on which they could be based.

(4)

Design of the CONSUL Programming Language

Doug Baldwin
Cesar Augusto Quiroz Gonzalez
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 208
February 1987

Abstract

We study the problem of automatically exploiting parallelism in computer programs, with particular emphasis on linguistic barriers to parallelism detection. Although functional languages and Prolog have many desirable characteristics in this respect, we find that they are not entirely ideal. We therefore offer constraint-based programming as a generalization of logic programming. By virtue of the more flexible ways in which they allow relations to be defined, constraint languages support more natural descriptions of potentially parallel algorithms than do existing logic or functional languages. We introduce a prototype constraint language called CONSUL, which demonstrates features that we feel make constraint languages well-suited for general-purpose programming of multi-processors. The extra expressiveness of constraint languages comes at a price, namely that satisfaction of general constraints can be much more difficult than satisfaction of predicates for a language like Prolog. Nonetheless, we believe that effective compilers for constraint languages can be built, and we outline some ideas on which they could be based.

This research was supported in part by the U.S. Army Engineering Topographic Laboratories under Contract DACA76-85-C-0001 and in part by the National Science Foundation under CER Grant DCR-8320136. The Xerox Corporation University Grants Program provided the equipment used in preparing the paper.

Contents

1 Introduction

Parallel computation is an area in which software technology lags considerably behind hardware technology. The need for parallel computing in a number of applications (e.g., scientific computing, machine vision, artificial intelligence) is unquestioned, and computers with hundreds of processors are now readily available (for instance, the Butterfly™ [2] or the many derivatives of the Cosmic Cube [24]). However, these machines are programmed in essentially the same way as existing sequential machines. The best available parallel programming languages are variants of standard sequential languages, with extensions to let the programmer explicitly divide a program into tasks and pass information between those tasks. Although designers of these languages claim that they are no harder to use than conventional sequential ones [13, 15], programmers still face the problem of figuring out how to partition their application into pieces in addition to the usual problem of translating it into a program. An appealing alternative is to leave partitioning of programs to compilers. Since it hides partitioning problems from programmers, this approach should make parallel computers much more usable than they are now.[1]

Unfortunately, progress on automatic parallelization has been extremely slow. The key problem is that of recognizing data dependencies, i.e., recognizing when two operations must be done sequentially because one produces or destroys a value that the other needs. Generally, any operations that are not explicitly serialized by data dependencies can be done in parallel. Comprehensive detection of data dependencies in traditional imperative languages is impossible, because of the use of side effects to maintain program state and the presence of aliasing (the possibility that several lexically distinct names actually refer to the same piece of state information). Declarative languages offer much more promise for automatic detection of data dependencies. The distinguishing characteristic of these languages is that their programs are descriptions of the features that make a result "correct" rather than of the detailed computations that produce that result. Declarative languages generally do not allow side effects, and thus have the so-called "single assignment property": every variable has exactly one value, defined in exactly one place. This property can make detection of data dependencies trivial — they simply exist between the definition of each variable and all of its uses, and nowhere else. If all else fails, data dependencies can be detected at run time by testing to see whether a variable is defined before using it (assuming that undefined variables can be marked somehow). Furthermore, the absence of side effects usually means that the semantics of declarative languages have a clear correspondence to well understood mathematics. From our point of view as compiler writers, this feature is desirable because it makes it

---

"Butterfly" is a trademark of BBN Laboratories, Inc.

[1] Of course, there will always be cases in which demands for extreme efficiency require manual parallelization. There is an exact analogy here to assembly languages versus high-level languages for fast, compact code: a few programs must be written in the lowest-level language possible for efficiency, but considerations of programmer productivity dictate that most programs be written in much higher level languages.

easy to reason formally about the compilation process. We therefore believe that the right starting point for automatic parallelization is a declarative language with a clean, mathematically oriented semantics.

We have chosen *constraint languages* as the declarative languages with which to work. A constraint language is one in which programs consist of sets of relations between inputs, outputs, and (possibly) intermediate values, such that the relations hold if and only if the output values are correct for the inputs. Constraint languages and logic languages are thus nearly the same thing: any relation in a constraint program can be replaced by a predicate that tests whether that relation holds, and any predicate in a logic program defines a relation between its arguments. There is, however, an important but subtle distinction between constraint languages and current logic languages: A constraint language provides a richer set of primitive relations than do existing logic languages. We believe that doing so makes the expression of general algorithms and their potential parallelizations more natural. The cost of the richer set of primitives is that satisfying systems of constraints is much harder than satisfying systems of predicates in existing logic languages. These points are discussed in detail in Section 3. Like logic languages, constraint languages do not distinguish between inputs and outputs, either of programs as a whole or of individual relations. This feature makes constraint languages more expressive than many other kinds of declarative language, specifically functional and data flow languages.

There are many different kinds of parallel computer and many different applications of parallel computing. Our research is aimed specifically at a certain class of MIMD (multiple instruction stream, multiple data stream) architectures and general purpose programming. The machines we are interested in consist of a few hundred processors, each capable of carrying out substantial computations independently of the others. Communication between processors is moderately expensive relative to computation (i.e., takes on the order of a few to several hundred times longer than a local memory reference). The Butterfly [2] is a good example of this kind of machine. This class of machines is of interest because it constitutes the current generation of general-purpose parallel computers. The vague term "general purpose programming" means that we are not designing our languages to the specific requirements of some narrowly defined class of applications. Instead, we hope to repeat the history of languages like C or Lisp (to take two of many examples), which, although perhaps designed for a particular use, have ended up being used in almost every kind of application. Our first constraint language is intended to fill roughly the niche on parallel machines that Pascal fills on sequential ones, namely laboratory development of non-trivial programs for a variety of applications. This goal reflects our research interests, and does not imply any feeling that constraint languages cannot be applied to other areas. In fact, if our first language is successful, we hope that later ones will be designed for real-world, production-quality, programming.

The ultimate goal of our research is to show that constraint languages are a practical tool for the kinds of programming described above. Achieving this goal

requires solving two key problems. The first is to determine the features that a general purpose constraint language should have; the second is to show that such a language can make effective use of a parallel computer. We have defined a language called CONSUL that we believe addresses the first problem. We are now conducting a series of experiments intended to test CONSUL on a variety of programs and to characterize the parallelism that it makes available in each. These experiments will (we hope) support our contention that CONSUL is suitable for general purpose programming, and will direct us to the richest sources of parallelism in the language. A later phase of the CONSUL project will address the second problem by trying to develop compilers that can exploit this parallelism on a real multi-processor (the University of Rochester's Butterfly). This paper argues that yet another programming language really is needed, describes the features of CONSUL that should make it easy to parallelize, and discusses the status of our experiments with it.

2 Previous Work, or, Why Yet Another Programming Language

As mentioned in the introduction, we see little hope for effective automatic parallelization of traditional imperative programming languages because of the presence of side effects and aliasing. Systems that do attempt to parallelize imperative languages (for example [11, 23]) usually work well only for particularly regular code (i.e., numeric routines), or specific machine architectures, or both. These systems thus do not satisfy our interest in general purpose programming, and often do not work for the architectures in which we are interested.

Although it is generally easier to parallelize declarative languages than imperative ones, all of the declarative programming styles proposed to date have drawbacks. One of the first declarative styles to receive widespread attention from the parallel programming community was functional programming [3]. Functional languages treat programs as mathematical functions, generally written as compositions of simpler functions. The use of higher-order functions as combining and control forms elegantly unifies computation, control, and communication into a single formalism. The fundamental source of parallelism in a functional language is concurrent evaluation of a function's arguments. However, argument parallelism is only one of several sources of parallelism in a program, and the others are generally missed by functional languages. The most important missed form of parallelism is data parallelism (simultaneously performing some computation on all elements of a data structure). The sequential replacement for data parallelism is iteration over the data structure, and most parallelizing compilers try to replicate loop bodies for parallel execution. In a functional language however, where iteration is implemented as recursion, this possibility is often masked by dependencies between a function's inputs and the arguments it passes to recursive invocations of itself. The usual solution to this problem is to introduce mapping functions into the language, i.e., functions that explicitly apply a second function to all elements of a list or other data structure. Unfortunately this approach has several drawbacks, specifically its

restriction to those data structures for which mapping functions are defined in the first place, an inability to handle cases in which the obvious data parallelism is over an *output* of the computation rather than over its inputs, and an inability to handle cases in which the goal of the computation is to produce some summary or aggregate of the inputs (for example, summing the numbers in an array).[2] In all cases where mapping functions cannot be used, the user of a functional language must fall back on recursion, with the extra data dependencies that it introduces. For the same reason, data parallelism is also hard to express in classes closely related to the functional languages, for example data flow languages [1] and equational languages [18]. Because real programs typically work on large data sets, data parallelism offers an important opportunity for massive parallelization. The inability to deal well with it is thus a serious limitation on the amount of parallelism that can be extracted from functional languages and their relatives.

More recently, logic languages have been intensively studied as sources of parallelism. Two promising sources of parallelism in a logic language are AND parallelism (concurrent evaluation of the conjuncts in a clause) and OR parallelism (concurrent evaluation of alternative clauses in a procedure). Furthermore, because logic languages allow systems of predicates to have multiple solutions and do not syntactically distinguish inputs from outputs, data parallelism is easier to express than in other declarative languages. We will return to this idea in the next section, when we discuss data parallelism in CONSUL. These advantages not withstanding, progress on parallelizing logic languages has been disappointing. One major problem is that there are many extremely common facilities that cannot be implemented in a practical form within the predicate calculus formalism of logic programming. Examples include arithmetic, input and output, et cetera. Existing logic languages provide these facilities, but as "extra-logical" constructs; because they are extra-logical they do not share fully in the benefits of logic programming for parallelization. Another difficulty seems to be that logic programming is virtually synonymous with Prolog [9], and Prolog is a very difficult language to parallelize. The search strategy most often used to generate results (backtracking) is inherently sequential, and a strict left-to-right evaluation order is guaranteed. The required order of evaluation combines with the fact that all clauses share a single data base but may use different names for its elements to make the problems of data dependency analysis and aliasing at least as severe in Prolog as in any imperative language. In order to get around these problems, a number of parallel variants of Prolog have been proposed [8, 25]. These languages accomodate parallelism via features for synchronizing producers and consumers of values, ways of committing the program to certain decisions instead of being able to backtrack out of them, et cetera. Effective use of these features requires users to be aware of the potential parallelism in their programs, and forces users to understand procedural aspects of program execution that ideal logic languages would abstract away.

[2] Readers interested in a more detailed discussion of these points are referred to [4].

Despite the problems with Prolog and its variants, the general promise of logic languages for parallel programming encourages us to look at closely related languages. Constraint languages, being generalizations of logic languages, are an excellent choice. The seminal work on constraint-based programming languages is Steele's dissertation [26]. Prior to Steele's work, constraint satisfaction had been used as a component in other systems [5, 27], but never as the basic computational model in a programming language. Steele's work is important for demonstrating a heuristic for satisfying constraints that should be easy to implement on either sequential or parallel computers (Steele only did sequential implementations), and that usually finds solutions in practical amounts of time. Unfortunately, as Steele himself points out, his language is not sufficiently developed to be "general purpose". One of the major difficulties is that there are no provisions for describing computations that depend on internal state. Although surprisingly sophisticated problems can be solved without explicit reference to state (for example, Steele describes a scheme for solving the N queens problem in his language), we believe state is vital in many real applications. For example, editors must apply each command to the text resulting from the previous ones, data bases without state are simply meaningless, and so forth. The remainder of this paper describes a new constraint language called CONSUL, which we have designed in an effort to make constraint-based programming truly general purpose.

3 The CONSUL Language

The formal foundation for CONSUL is axiomatic set theory. Thus the fundamental data type is the set, and the fundamental operators are the logical connectives and quantifiers. However, a number of abstractions are built in to the language to make it more palatable than raw set theory to programmers. In particular, the built-in data types include familiar ones such as sequences, integers, characters, et cetera. Each of these types can be given a set-theoretic definition, but programmers generally need not be aware of it. One consequence of the formal basis of CONSUL that can be important to programmers, however, is that relations, being sets, can be treated as data, and vice versa. This feature allows the language to include higher-order relations in a natural way. Each built-in data type is associated with built-in relations that correspond to common operations for that type. Thus CONSUL provides simple comparisons, arithmetic relations between integers, and so forth as language "primitives". Again, the fact that these operations are not really primitive to the underlying set theory is invisible to users. The built-in relations can be composed into more complex ones using the logical connectives "and", "or", and "not", with their standard meanings, and the quantifiers "for all" and "there exists". An example and discussion of the key features of CONSUL appears below.

With its basis in set theory, CONSUL is superficially similar to SETL [10]. The two languages differ greatly in orientation however: SETL is an attempt to abstract data structure definition out of traditional imperative languages, whereas CONSUL is an attempt to abstract explicit descriptions of concurrency out of programming by

using a non-traditional declarative language. CONSUL also has certain similarities to Crystal [7]: Both are motivated by a belief that a clean mathematical notation is an appropriate way of expressing parallel algorithms, and both use sets as the basic aggregate data type. Crystal, however, is a functional language (i.e., the basic act in executing a program is applying a function to some inputs to produce some outputs), whereas CONSUL is a constraint language (the basic action is solving an equation).

3.1 An Example

Figure 1 shows a simple lexical analyzer written in CONSUL. The (nominal) input to this program is a sequence of characters; the output is a sequence of numeric codes (tokens) representing the words in the input. Input words are assumed to come from the vocabulary "red", "blue", "green", "black", and "yellow", with words separated by any number of spaces or new-line characters. The last character in the input is assumed to be either a space or new-line. The output encodes "red" by the integer 1, "blue" by 2, "green" by 3, "black" by 4, and "yellow" by 5. This example is thus a simple paradigm for the front-end to any command-driven program (e.g., a shell, an editor, et cetera). The classical approach to this kind of analysis is to pass the inputs through a state machine, with output tokens being determined by the machine's state at the end of each word. Note, however, that a great deal of parallelism can be exploited by searching the input for word boundaries and passing each word to the state machine as soon as its end is found (concurrent with the rest of the search and processing of other words). This is the approach taken in the CONSUL solution, where relation "Number-Delimiter-Groups" defines the beginning of each word and "FSM" describes the state machine. States are encoded as integers in such a way that tokens are just final state codes.

CONSUL's syntax is based on that of Lisp. A constraint in CONSUL is an S-expression. The first element of the expression names a relation that is to hold between the remaining elements.[3] Arguments appear in a "natural" (at least to the authors) order. Thus, for example, `(plus x y z)` means $x = y + z$, `(greater a b)` means $a > b$, `(elt x a i)` means $x = a[i]$, et cetera. Anonymous constraints are constructed by "rho", in analogy to Lisp's "lambda" for constructing anonymous functions. Named constraints can be defined with "defrel". A convention for nesting constraints using "%" to share values between inner and outer ones has been borrowed from Steele [26]. Thus the constraint

```
(less I (size % Out))
```

is identical to

```
(exists ((Temp integer))
   (and (size Temp Out)
        (less I Temp)))
```

[3] As a point of terminology, we use the term "relation" to denote the mathematical concept, and "constraint" to denote its syntactic representation in CONSUL.

```
;;; Delim - Holds if and only if C is a delimiter character.
(defrel Delim (C)  (member C {set #\space #\newline}))

;;; Number-Delimiter-Groups - Number of groups of adjacent delimiters between
;;; In[0] and In[Pos] is Count, and In[Pos] is not in the middle of a word.
(defrel Number-Delimiter-Groups (Count In Pos)
   (exists ((Ends {power-set integer}))
      (and (or (equal Pos 0)
               (Delim (elt % In Pos)))
           (equal Ends {subset integer
                               (rho (I)
                                 (and (greater I 0)
                                      (less I (plus % Pos 1))
                                      (Delim (elt % In I))
                                      (not (Delim (elt % In (minus % I 1))))))})
           (size Count Ends))))

;;; Final-State - Holds iff State is a final state of the state machine.
(defrel Final-State (State)  (member State {set 1 2 3 4 5}))

;;; Trans - Holds when FSM can go from state Now to state Next on input C.
(defrel Trans (Now C Next)
   {set (0 #\space 0) (0 #\newline 0)          ; Ignore leading space
        (0 #\r 6) (6 #\e 7) (7 #\d 1)          ; Recognize "red" as 1
        ...})

;;; FSM - State machine yields Token if started at In[Start] in state State.
(defrel FSM (Start In Token State)
   (or (and (Final-State State)
            (equal Token State))
       (and (not (Final-State State))
            (FSM (plus % Start 1) In Token (Trans State (elt % In Start) %)))))

;;; The main analyzer:
(and (size (Number-Delimiter-Groups % In (minus % (size % In) 1)) Out)
     (forall ((Token Out))
         (exists ((Pos integer)
                  (Start integer))
             (and (index Pos Token)
                  (Number-Delimiter-Groups Pos In Start)
                  (FSM Start In (datum % Token) 0)))))
```

Figure 1: A Lexical Analyzer Written in CONSUL

This nesting is purely to save programmers the nuisance of creating large numbers of temporary names, and has no effect on the semantics of CONSUL. Forms enclosed in braces ("{" and "}") are set constructors. These forms are not true constraints, but are more like macros for constructing sets. They are discussed in more detail below. Some other important features of CONSUL are as follows:

Program structure: The key forms for building CONSUL programs are the connectives "and", "or", and "not", and the quantifiers "forall" and "exists". Each of these constructs has its normal logical meaning. Note that the connectives do not imply any order in which the connected clauses will be solved. AND and OR parallelism can be exploited by CONSUL as by other logic languages. The quantifiers introduce one or more new variables, whose scope is restricted to the quantifier's body. Each variable denotes a value that either ranges over (for "forall") or must lie in (for "exists") a designated set. The syntax for both quantifiers is

(quantifier-name ((var_1 set_1)
...
(var_n set_n))
body)

Note that the scope of the names introduced by "forall" or "exists" does not include the definitions of other variables introduced by the same quantifier. The major relations in the lexical analyzer (for instance, "Number-Delimiter-Groups", "FSM", and the main analyzer) demonstrate typical CONSUL structure. As suggested by these examples, "and" and "or" can define blocks of constraints, and are often used to define the body of a quantifier. "FSM" demonstrates the CONSUL idiom corresponding to conditionals in imperative languages. The simplest form of this idiom corresponds to the if-then-else construct, and looks like

(or (and ⟨*condition*⟩
⟨*then−body*⟩)
(and (not ⟨*condition*⟩)
⟨*else−body*⟩))

This construct generalizes easily to forms that correspond to Lisp's "cond". As seen in the example, "exists" is mostly used to define local variables (for example, "Ends" in "Number-Delimiter-Groups" and "Start" and "Pos" in the main analyzer). "Forall" can be viewed as a way of mapping relations over sets, and is discussed in the next paragraph.

Data Parallelism: CONSUL shares two properties with logic languages that help in the expression and detection of data parallelism. First, relations do not distinguish inputs from outputs, so mapping operators can be used to describe more kinds of data parallelism than in functional languages and their relatives. In particular, it is as easy to define a computation as a mapping of some relation over the final result as to define the computation as a mapping over inputs (although in doing so one must be careful not to write constraints that are trivially solved

by the empty set. i.e., no result at all). In CONSUL, mappings of any sort are written using "forall", which maps a relation over one or more sets. The main body of the lexical analyzer demonstrates "forall". Since the quantified variable in this "forall" ranges over output tokens, this example also demonstrates mappings over outputs. An English specification for the main body could be "There is an output token for every input word, and every token corresponds to the final state of the state machine when it is run on the corresponding word". The first clause of this specification is expressed in CONSUL by the "size" sub-form; the second by the "forall". Note that in describing the analysis in terms of "every token", the English leads naturally (at least to our way of thinking) to data parallelism over the outputs of the program. We do not see any equally simple specification that leads as naturally to data parallelism over inputs. Thus the example shows that the insensitivity of logic and constraint languages to distinctions between input and output really can be a significant help in writing parallel programs.

Even the relatively powerful kinds of mapping allowed by constraint and logic languages do not allow all forms of data parallelism to be expressed. Those cases that cannot be described by mapping must still be described recursively. As discussed earlier, recursion introduces data dependencies that can obscure data parallelism. The second way in which constraint (and logic) languages support data parallelism is by helping distinguish these spurious dependencies from dependencies that are important parts of a program. The problem with data dependencies in most declarative languages is that by making one value depend in a specific way on another they force a computation to be done in a specific order. Parallelism, on the other hand, requires that the steps of a computation can be done in any order. In CONSUL one can write constraints that have multiple solutions. If the values involved in a data dependency are solutions to such constraints, then each possible solution corresponds to a different data dependency, and hence to a different execution order. Thus the possibility of multiple execution orders (in other words, of parallelism) is expressed in recursive CONSUL programs by giving deliberately imprecise (although still correct) definitions of the arguments to the recursion. For example, a constraint "Total" that requires "Tot" to be the total of the numbers in "Set" can be defined as follows:

```
(defrel Total (Tot Set)
   (exists ((X Set))
       (or (and (size 1 Set)
                (equal Tot X))
           (and (greater (size % Set) 1)
                (plus Tot
                      X
                      (Total % (set-difference % Set {set X})))))))
```

This definition can be read as "The total of a one-element set is the element; the total of a larger set is computed by removing some element from the set, totalling the remaining elements, and then adding the removed element to the sub-total".

The key point is that *any* element of the set can be removed, and this is precisely what the CONSUL program says — "X" need only be *some* element of "Set", not one specific element. We hope that compilers can be written that detect at least some of these imprecise data dependencies and remove them for parallel execution.

Sequences: One of the most important kinds of set in CONSUL is the sequence or tuple. The importance of sequences comes from the fact that they are the only ordered structures in CONSUL. The lack of an ordering in CONSUL's other data and control structures is a deliberate effort to give compilers maximum flexibility in parallelization. However, the idea of an ordering is essential to some applications. For instance, consider the lexical analyzer from Figure 1. The ordering of words in its input defines an ordering of tokens in its output, and the analyzer would be useless if tokens appeared in some other order. Typically ordering arises in these applications because of the way in which they interact with users or some other external system. Internally, even elements of ordered data structures can often be processed in parallel. CONSUL reflects this observation by treating sequences as special kinds of set. In other words, all of the sources of parallelism discussed for sets also apply to sequences, yet sequences still provide a way to describe essential orderings. The lexical analyzer's data parallel generation of tokens is a good example. (Actually if one tries hard enough one can write sequential code using sequences — for an example, look ahead to Figure 2 and note the data dependency between successive elements of "States".)

Inputs and outputs of CONSUL programs are modelled as sequences, with many of the properties of streams in functional languages. In the version of CONSUL used in the lexical analyzer, "In" names the single input sequence and "Out" the single output sequence. Of course more mature versions will include multiple inputs and outputs and ways of binding them to external files. We assume that the external representation of a CONSUL I/O sequence has a natural ordering that corresponds to the index ordering within the program. Examples are the temporal order in which characters are typed at a keyboard or the spatial order of records in a file.

Sequences need not be homogeneous, and so can have elaborate internal structures. This feature means that CONSUL sequences are equivalent to lists in languages like Lisp or Prolog. Just as in Lisp, a CONSUL program can be represented by a CONSUL sequence. When reading the CONSUL examples in this paper, parentheses should be thought of as sequence delimiters.

For all the importance of sequences, the precise way in which CONSUL will let programmers "see" them as sets has yet to be determined. For now it seems desirable for a sequence to be viewed as a set of ordered pairs, with each pair representing one element of the sequence.[4] Each pair contains an index, giving the position of the element in the sequence, and a datum indicating the value of

[4] Note that we say "viewed" rather than "represented", since (as with most other primitive data types) the way in which programmers think about sequences should be very different from the way implementations actually store them.

the element. In this paper we use the constraint `(index i x)` to mean that "i" is the index component of ordered pair "x" and `(datum d x)` to mean that "d" is the datum component of "x". The main advantage of this view is that it allows sequences to be used interchangeably with sets in such forms as "size", "forall", et cetera. The disadvantage is that the ordered pairs used to represent elements of a sequence will not look like 2-element sequences, even though ordered pairs and 2-element sequences are mathematically the same thing (otherwise 2-element sequences would be defined in terms of themselves). This distinction makes the formal description of CONSUL slightly less elegant than we would like, but does not pose serious pragmatic problems. It is also clear that the proposed view of sequences is just a special case of a much more general associative structure. As we gain experience with CONSUL, we may decide to replace sequences as primitives with this more general structure.

Set constructors: Since the set is CONSUL's primary data structure, the language provides a powerful collection of primitives for describing sets. These set constructors include most of the common mathematical operations on sets, including power set, cross product, et cetera. Figure 1 demonstrates several set constructors, including "set" (builds a finite set from a list of its arguments), "subset" (`{subset s c}` is the set of elements of "s" for which constraint "c" is satisfied), and "power-set" (builds the power set of its argument). Several simple but important sets are available as CONSUL primitives, and can be used with the constructors to define more complicated sets. For example, the CONSUL primitive "integer" stands for the set of all integers, and "empty" represents the empty set. Set constructors are human-readable representations of the data structure used by CONSUL to represent a set, and so are not constraints. (But note that a set can be represented in part by a constraint that its elements satisfy, so the values referenced in a set construction need not be compile-time constants. This feature is used to advantage to define the value of "Ends" in "Number-Delimiter-Groups" in the lexical analyzer.) The reason for distinguishing set constructors from constraints in CONSUL has to do with boot-strapping set definitions — if set constructors were constraints it would be necessary to name the constructed sets, but names must be introduced by "exists" or "forall", which require a set from which each name can take values, which in turn brings us back to the problem of defining sets.

Data types: The set of possible values that "exists" and "forall" associate with a newly declared variable can be thought of as the variable's type. For example, the "exists" in "Number-Delimiter-Groups" declares "Ends" to be of type "set of integers". In order to simplify the definition of complicated data types (among other things), CONSUL provides a "define" form that allows programmers to name arbitrarily constructed sets. Using "define", the example from the lexical analyzer could have been written

```
(define Integer-Set {power-set integer})
...
(exists ((Ends Integer-Set))
    ...)
```

```
;;; FSM - Holds if and only if Token is the final state reached by running
;;; the word recognizer on characters from In starting at position Start.
(defrel FSM (In Start Token)
   (exists ((States {sequence integer}))
      (and (equal (elt % States 0) 0)
           (Final-State (elt % States (minus % (size % States) 1)))
           (equal Token (elt % States (minus % (size % States) 1)))
           (forall ((I {subset integer
                               (rho (I)
                                 (and (greater I -1)
                                      (less I (size % States))))})))
              (Trans (elt % States I)
                     (elt % In (plus % Start I))
                     (elt % States (plus % I 1)))))))
```

Figure 2: Non-recursive State Machine for Lexical Analyzer

User-defined relations: CONSUL programmers can define new constraints to represent arbitrary relations. "Defrel" and "rho" are used for this purpose, depending on whether one wants to name the new relation or not. Most top-level forms in the lexical analyzer are examples of "defrel"; a use of "rho" can be seen in "Number-Delimiter-Groups". It is most common to define new constraints as a block of code with parameters, analogous to function definitions in other languages. However, the set-theoretical semantics of CONSUL mean that a constraint is really just a representation for a (possibly infinite) set of tuples. Thus one can define finite constraints by explicitly listing the tuples that belong to them (see "Trans" in the lexical analyzer). Applying (i.e., "calling") a constraint means finding an assignment of values to its arguments such that when these values are grouped into a tuple in the order in which the arguments are given to the constraint, that tuple is an element of the set represented by the constraint. CONSUL is lexically scoped, thus providing an interpretation for free variables in the body of a constraint. The "FSM" relation in the lexical analyzer shows that constraints can be defined recursively. Such descriptions can be natural reflections of how a programmer thinks about a relation, particularly if the relation is defined in terms of internal state variables. However, CONSUL often allows more declarative definitions also. For example, the recursion in "FSM" could be eliminated by asserting that there exists a sequence of states starting with 0 (the start state) and ending with a final state, such that "Trans" holds between pairs of adjacent states and the corresponding input character. This version of "FSM" is shown in Figure 2.

3.2 A Possible Execution Model for CONSUL

CONSUL has two characteristics that make it particularly attractive to us as a parallel programming language. The first is the fact that every variable in a CONSUL program represents exactly one value, i.e., CONSUL is a single-assignment

language.[5] As noted earlier, the single-assignment property makes detection of data dependencies much easier than it would be otherwise. Data dependency detection in CONSUL is not quite as easy as in some other single-assignment languages (for instance functional languages), because the value of a CONSUL variable can be defined jointly by several constraints. Thus there may be times during the solution of a system of constraints when a variable's value is only partially known (for example, it might be known to be an integer greater than zero, but not which such integer). We believe we can distinguish partially known values from fully known ones, and so they will not pose a serious problem for CONSUL translators. The basic idea is to bind variables to *sets of possible values* rather than to just one value. A variable whose value is known precisely is then represented by a set containing a single member; a partially known value is represented by a set with multiple members. When a variable is bound to a set with only one member we say that it is *precisely* bound. A variable that is not precisely bound is *imprecisely* bound. Since any CONSUL implementation must have ways of representing sets anyhow, this way of binding variables to values should be fairly easy to implement.

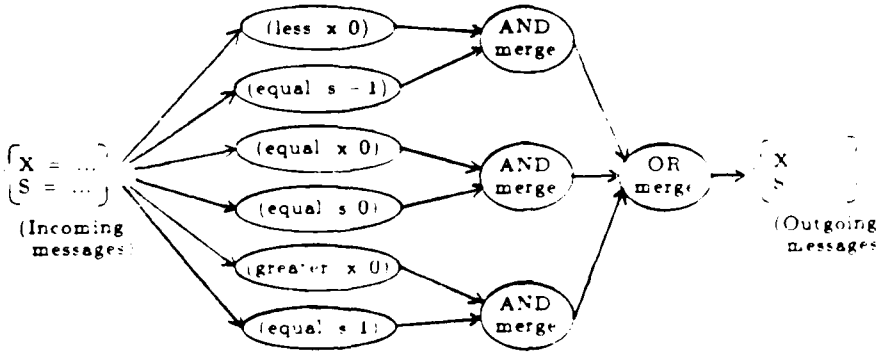
The second advantage of CONSUL for parallelization is that there appears to be a straightforward way of partitioning CONSUL programs into concurrent processes. (But the straightforward partitioning is not necessarily a very good one. Ways of improving it are suggested below.) Note that the present discussion of this strategy is preliminary, and may change with time. The basic idea is that each constraint in a program is solved by a separate process, with processes communicating by message passing. Messages contain bindings of variables to (sets of) values. Each message contains a complete set of bindings, i.e., bindings for all variables accessible to the constraint that generated it. Special processes corresponding to entries into new scopes and exits from old ones inject new variables into messages and filter out those that are no longer visible. Whenever a process receives a message it immediately tries to solve its constraint for the received bindings. If (and only if) a solution is found, the process then sends messages of its own containing updated bindings. An example of this strategy is shown in Figure 3.

The constraint shown in Figure 3 just requires that "S" have a value of -1, 0, or 1, corresponding to the sign of "X". Each of the base constraints in the CONSUL code corresponds to a process (denoted by ovals in the diagram). "And" and "or" forms correspond to processes that merge streams of messages. An "or" merge process simply receives messages from any of its possible senders and relays them to its receiver(s). OR parallelism is exploited, since everything that can send to an "or" merge is a separate process that runs concurrently with other senders to the "or". The fact that processes do not share the data structures describing bindings (i.e., the use of message passing instead of shared memory for interprocess communication) is what enables us to use such a simple implementation of OR parallelism. If processes did share bindings then there would be a danger of logically

[5] The quantified variable in "forall", which ranges over a set of values, is considered to be a family of variables, one per element of the set.

```
(defrel Sign (X S)                          ; S is the sign of X
   (or (and (less X 0)
            (equal S -1))
       (and (equal X 0)
            (equal S 0))
       (and (greater X 0)
            (equal S 1))))
```

a: CONSUL Code



b: Process Structure

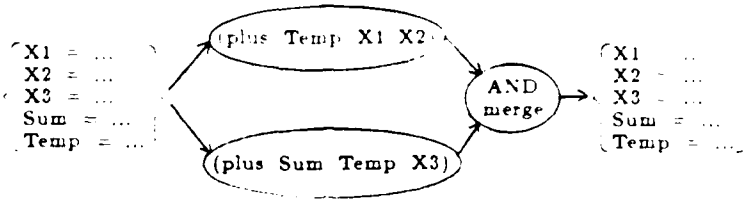
Figure 3: A Constraint and its Partitioning into Processes

independent alternatives making mutually inconsistent changes to the bindings. We avoid this problem by letting each message be a complete, self-contained solution to a set of constraints, and by placing different solutions in different messages. AND parallelism can also be exploited, using "and" merge processes. An "and" merge process is considerably more complicated than an "or" merge however. An "and" merge waits until it has received one or more messages from each of its possible senders, and then combines the bindings contained in these messages into consistent sets of bindings to be placed in output messages. Note that this way of mapping "and" forms into process structures is only one of several possibilities. Another option, that may be much more efficient in some cases, is discussed in the next paragraph.

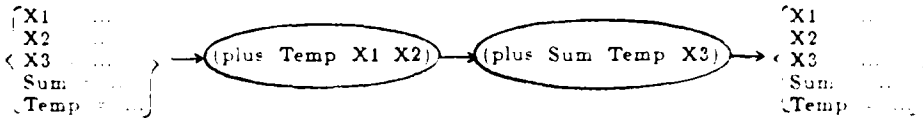
Not surprisingly, the AND parallel process structure illustrated in Figure 3 does not work very well if the processes corresponding to the body of the "and" do not have much inherent parallelism. Specifically, if there are data dependencies between sub-forms of an "and", the "and" merge process may have to wait a very long time before it can make a consistent set of bindings from the messages it receives. As an example, consider the constraint shown in Figure 4a. This constraint uses two "plus" forms to force "Sum" to be the sum of "X1", "X2", and "X3". Assuming that "X1", "X2", and "X3" are precisely bound, then the first "plus" constraint defines

```
(defrel Sum-Of-3 (Sum X1 X2 X3)
   (exists ((Temp integer))
      (and (plus Temp X1 X2)
           (plus Sum Temp X3))))
```

a: CONSUL Code



b: Process Structure Using "and" Merges



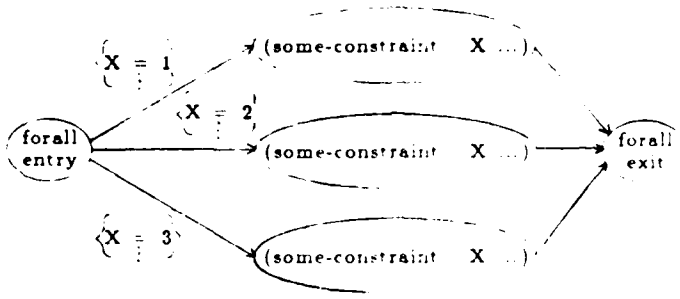
c: Process Structure Respecting Data Dependencies

Figure 4: Handling Data Dependencies in "And" Forms

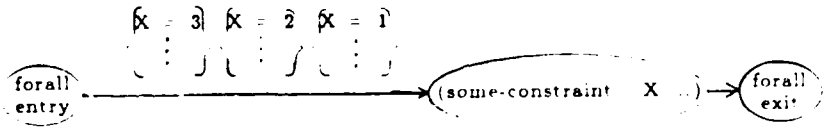
a unique precise binding for "Temp". Given this binding, only one precise binding of "Sum" is possible. Without a precise binding of "Temp", however, the second "plus" constraint can produce an infinite number of consistent precise bindings for "Temp" and "Sum" jointly. If an "and" merge is used to implement "Sum-Of-3", as shown in Figure 4b, that is exactly what will happen: The process corresponding to the first "plus" will send a single message to the "and" merge; the process corresponding to the second "plus" will send an infinite number of messages. The "and" merge must select from this second stream the one message containing the binding for "Temp" that is consistent with the first "plus". A much more efficient implementation would eliminate the merge and just have the first "plus" send its message directly to the second, which could immediately compute the one successful binding of "Sum". This solution is shown in Figure 4c. Note that this solution has no parallelism, but this follows directly from the absence of useful parallelism in the code from which it was generated. As a general rule, data independent sub-forms of an "and" can be solved concurrently, with results combined in an "and" merge, whereas data dependent sub-forms are better solved serially. In the latter case the partitioning strategy can be improved by having a single process solve all

```
(forall ((X {set 1 2 3}))
   (some-constraint X ...))
```

a: CONSUL Code



b: Replicating the Body of the Form



c: Pipelining

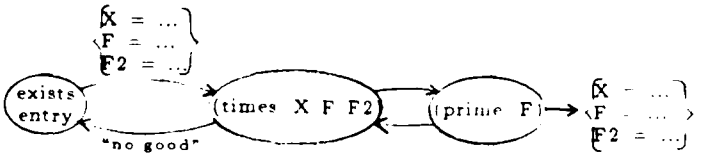
Figure 5: Partitioning a Data Parallel Form

the serialized constraints, since there is no parallelism to be exploited by multiple processes.

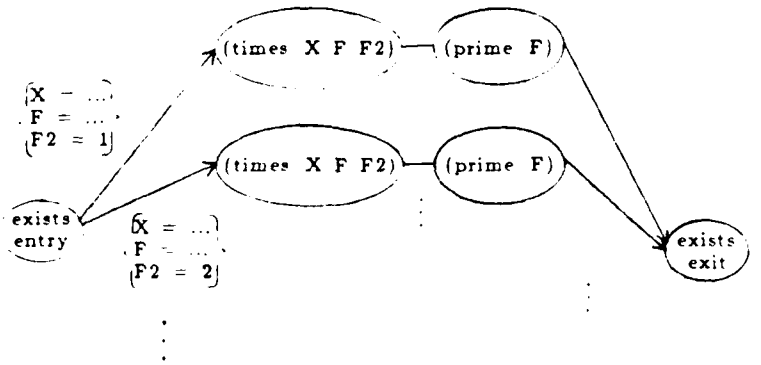
Data parallelism can be implemented in our execution model by replicating the data parallel form once for each data value. An example of this replication appears in Figure 5b. The "forall entry" and "forall exit" processes in this example delimit the scope of the quantified variable introduced by the "forall". The "forall entry" distributes incoming messages to each replica of the body of the "forall", supplementing each outgoing message with a different precise binding for the quantified variable. The "forall exit" acts much like an "and" merge. It waits until each replica of the body has succeeded at least once, and then tries to combine the received bindings into one or more outgoing messages. Another approach to data parallelism is to pipeline it: The entry process sequentially sends messages containing different bindings for the quantified variable into the body. Figure 5c demonstrates this approach. Replicating the body should provide more parallelism than pipelining (as long as the data structure involved is large enough), but may be harder to implement (a very large number of replicas may swamp the target machine, and in some cases it will be hard to tell how many replicas are needed).

```
(defrel Prime-Factor (X F)                    ;F is a prime factor of X
   (exists ((F2 integer))
      (and (times X F F2)
           (prime F))))
```

a: CONSUL Code



b: Message-Based Backtracking



c: Data-Parallel Search

Figure 6: Two Approaches to Nondetermistic Computation

Both approaches should extend to data parallelism arising from forms other than "forall".

One of the strengths of constraint languages is their ability to express nondeterministic computations. Our execution model supports two ways of implementing nondeterminism. One is to allow processes to respond to receipt of a set of bindings with a "no good" message that causes the sender to backtrack. The other is to treat nondeterminism as a data parallel search over the set of possible solutions. In this approach the body of the nondeterministic computation is replicated (or alternatively pipelined) once for each possible solution, with an entry process sending different possibilities to each replica. An exit process acts like an "or" merge, forwarding any incoming message after removing local variables from it. Both approaches are illustrated in Figure 6. As usual, the different approaches have complementary advantages. Message-based backtracking serializes computation, but involves relatively few processes. As with serialized sub-forms of an "and", one can improve partitioning by solving systems of constraints that are expected to

backtrack in a single process. Parallel searching is highly parallel, but may involve inordinately large numbers of processes, many of them "wasted" in the sense that they will ultimately fail to contribute solutions.

The main problem with the execution model described here is that the individual processes are likely to be small and the messages they exchange large. In other words, overhead of process creation and communication may eliminate much of the advantage of parallel execution. The solution to this problem is to give each process more work to do (thus increasing its ratio of useful computing time to overhead time) and to reduce the size and number of messages sent. Ways of increasing process size by coalescing processes that would be serialized anyhow by data dependencies or backtracking have already been pointed out. Many of the entry, exit, and merge processes included in the above discussion can also be eliminated. Entry processes that do nothing but insert local variables into messages can be absorbed into the first processes that use the local variables, exit processes can be similarly absorbed into the last processes with access to locals, and "or" merges can be eliminated by having any process that would send a message to the merge instead send directly to the processes that would ordinarily receive from the merge. Finally, it may well be useful to combine even potentially concurrent constraint satisfactions into a single process just to balance the work done by each process against the overhead incurred by it. Various graph partitioning techniques (e.g., Kernighan and Lin's heuristic [20] or min-cut algorithms [12]) seem like promising candidates for this combining step. Note that all of these strategies for reducing the number of processes in a program also reduce the number of messages, since some communications that would have been via interprocess messages are now just interactions between different parts of a single process. It should also be possible to reduce the size of the messages that are sent. For instance, one way for process A to send environment E to process B is as the difference (i.e., set of variables with different bindings) between E and the previous environment sent from A to B. Other ways of reducing the overhead of message passing will be sought as compiler development begins.

There are a number of similarities between this execution model and other approaches to constraint satisfaction or parallel computation. Perhaps the most obvious similarity is between our model and the data flow model of parallel computation [1]. The main difference between our model and data flow is that we pass complete environments between processes rather than values of individual variables. Since each message is a complete environment, we do not need to synchronize a process's computation to the arrival of multiple values on different inputs to the process. Our model is also quite similar to Steele's approach to constraint satisfaction [26]. Again, the main difference is in passing complete environments rather than individual values. Passing complete environments simplifies synchronization again in this case, and also seems to simplify the handling of nondeterminism. Li describes a message-passing method for parallel execution of logic programs [21], but messages still contain individual values rather than complete environments (Li notes that passing complete environments is more expensive than passing single values — we agree, but feel that its benefits for synchronization and nondeterminism are worth

the cost). Li does describe an algorithm for merging results from multiple sources that may be applicable to the "and" merges in our model. Note, however, that our model has alternative implementations of "and" that do not require merging, and so we may be able to limit merging to situations in which much simpler merging algorithms can be used.

3.3 CONSUL and Logic Programming

The important difference between constraint languages (and hence CONSUL) and logic languages (i.e., Prolog) is the richer set of primitives present in a constraint language. In some cases, this richness has important advantages for parallel programming. For instance sets, as general, unordered data structures, seem to support data parallelism well. Sequences are another example. Although they can be defined using sets, the benefits of providing them as primitives (namely an I/O model and a general ordering mechanism) dictate that one do so. In other cases we give full relational definitions to primitives that logic languages treat as extra-logical. Common data types such as numbers or characters fall into this category. We feel that because all primitives are defined in the same formal framework, constraint language translators should face fewer special cases than those for logic languages, and the languages themselves should be easier to use.

Unfortunately, the presence of a uniform set of powerful primitives makes satisfying CONSUL constraints much harder than satisfying predicates in languages like Prolog. In particular, many of the primitives constrain two objects to be equal. Theorem proving (which is closely related to constraint satisfaction) in the presence of equality is the topic of intense research [14, 17, 19], but general procedures for doing it are not yet known. In addition to the problem of equality, satisfying CONSUL constraints is P-Space Hard, since CONSUL allows arbitrary nesting of universal and existential quantifiers. For these reasons, CONSUL implementations will be largely heuristic. The heuristic approach should not be a serious handicap however. Folk wisdom about Prolog suggests that most logic programs are written in simple styles that allow much more direct solution than admitted by theoretical complexity measures. The same should be true of CONSUL, suggesting that even theoretically very expensive solution strategies may perform well in practice, and that it may in many cases be possible to optimize an expensive search for a solution into a direct computation. Such optimizations can be based on symbolic simplification of constraints, using techniques available from the symbolic algebra literature [6]. One use of this approach has already been reported by Gosling [16], who claimed a substantial benefit even from trivial algebraic transformations. When all else fails, CONSUL translators should be able to recognize programs for which the available solution strategies may not work, and alert their users to the problem. Annotations can be added to CONSUL to let users help translators out of such situations. We expect, however, that these annotations will not need to be widely used.

Despite the differences, constraint and logic languages also have much in common. Many of the same sources of parallelism appear in both, and the key CONSUL control forms ("and", "or", and "forall") have simple implementations in Prolog.

These observations suggest that our work on parallelizing CONSUL should be important to work on parallelizing other logic languages and vice versa.

3.4 Experience with CONSUL and Status of the Project

At present (February 1987) we have a definition for a primitive dialect of CONSUL. We are working on writing a crude interpreter for this dialect as part of the parallelism measurements discussed below. This interpreter will support the semantics presented in this paper (although with help from users in satisfying constraints), but will differ slightly from this paper in syntax. In particular constraints cannot be nested using "%", and forms such as "defrel" or "member" that are really shorthand notations for awkward combinations of more primitive forms[6] are missing. One important role for the interpreter is to give us a concrete framework within which to write sample CONSUL programs. This exercise will provide some evidence to support (or disprove) the claim that CONSUL is suitable for general-purpose programming, and will help us design more user-friendly dialects of it. To date only a few programs have been written (e.g., the lexical analyzer presented above). We are also using the interpreter as a vehicle for testing implementation ideas that may be used by a full compiler, for example representations of sets and partially-defined values.

The main reason for writing the CONSUL interpreter is to allow us to measure the parallelism available in real CONSUL programs. To do this, the interpreter relies on its user to provide values satisfying each constraint. The interpreter proper notes the order in which the user satisfied constraints and which variables were read and defined for each. This information is written to a trace file, which is later compacted into a maximally parallel form by a compactor. The interpreter also handles routine book-keeping and execution modelling. The trace files generated by the interpreter contain complete information on the modes, values, and data dependencies encountered in actually executing a program. It is thus easy to compact them into the most parallel possible form, providing a measure of the parallelism that could have been exploited by the program. The granularity of compaction is the constraint, i.e., the compacted traces indicate which constraints could have been satisfied in parallel with which others. Because the traces are taken from programs as they run, the parallelism found by the compactor is "oracular" (see [22]) in the sense that a real compiler could fully exploit it only if it had perfect information about the object program's run-time behavior. The compactor also ignores communication and other overheads of real parallel execution. Our results thus indicate an upper bound on the parallelism that can be derived from CONSUL programs, but do not indicate how close a real compiler can come to that bound. The experiments are intended only to determine whether there really is substantial parallelism in CONSUL programs, and thus whether it is worthwhile to try to build compilers that can find it.

---

[6] "Defrel" is a special case of "define", and `(member x s)` is the same as the application `(s x)`.

CONSUL will continue to evolve as experience with it accumulates. It is already clear that implementing the full syntax from this paper will be a big step towards making CONSUL programs less verbose. A macro facility would also make the language much friendlier. Farther in the future, the need for and uses of annotations to guide compilers must be studied. Finally, there will doubtless be developments whose exact nature we cannot yet predict in areas such as new data types, interfacing CONSUL programs to their host operating systems, et cetera.

4 Summary and Conclusions

We have addressed the problem of automatically exploiting parallelism in computer programs, with particular emphasis on linguistic barriers to parallelism detection. Although functional and logic languages have many desirable characteristics in this respect, we find that they are not entirely ideal. We therefore offer constraint-based programming as a generalization of logic programming. By virtue of their richer sets of primitives and the more uniform ways in which they allow relations to be defined, constraint languages support more natural descriptions of potentially parallel algorithms than do existing logic or functional languages. This extra expressiveness comes at a price, however, namely that satisfaction of general constraints can be much more difficult than satisfaction of predicates for a language like Prolog. None the less, we believe that technology can be developed for building effective compilers for constraint languages — much of what is needed even exists now.

We have defined a prototype constraint language called CONSUL, which includes the features that we feel are necessary for general-purpose constraint-based programming of multi-processors. The key elements of CONSUL for parallel programming are the following:

- Data structures and control structures are unordered; AND and OR parallelism can be exploited.
- The availability of a powerful mapping operator and the ability to specify values imprecisely (with the imprecision possibly detectable by a compiler) support data parallelism.
- Sequences allow computations in which data must be ordered to be described, often without inhibitting parallelization.
- The language's main sources of parallelism correspond to a simple execution model that appears to be easy to implement on real multiprocessors.

A formal definition of CONSUL's semantics is possible based on axiomatic set theory.

The main contribution of our work so far is a prototype for a language that we (and others) can use as a base for research in general-purpose constraint-based parallel programming. Unlike most other languages designed for parallel programming, the parallelism in ours is to be detected by a compiler, not by the programmer. Although both constraints and sets have been used in other languages, ours is the first

(that we know of) to try to use the full language of set theory directly as a programming language. Future work will include justifying the claim that our language is really general purpose (and doubtless modifying it to make it more so), completing our measurements of the parallelism that it makes available, and trying to develop a practical compiler for it. The latter work is particularly important because of our undoubtedly controversial decision to emphasize clean, parallelizable semantics over general implementation algorithms. We hope that the CONSUL project will convincingly demonstrate that constraint languages are a sound foundation for powerful, convenient, parallel programming.

Acknowledgements

We wish to thank Jerry Feldman for helpful conversations during the course of this work and for his comments on earlier drafts of this paper. David Sher expressed a healthy skepticism that encouraged us to think about the problems of implementing CONSUL. A number of members of the USENET Prolog community helped by pointing out Prolog implementations of CONSUL's "forall".

References

[1] Ackerman, W. "Data Flow Languages". *Computer*, Feb. 1982. pp. 15-25.

[2] "Butterfly™ Parallel Processor Overview". BBN Laboratories Inc., June 1985.

[3] Backus, J. "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs". *Communications of the ACM*. Aug. 1978 (**21:8**). pp. 613–641.

[4] Baldwin, D. "Why We Can't Program Multiprocessors the Way We're Trying to Do It Now". Department of Computer Science, University of Rochester. In Preparation.

[5] Borning, A. "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory". *ACM Transactions on Programming Languages and Systems*, Oct. 1981 (**3:4**). pp. 353–387.

[6] Buchberger, B., G. Collins, and R. Loos (eds). *Computer Algebra: Symbolic and Algebraic Computation* (2nd edition). Vienna: Springer-Verlag, 1983.

[7] Chen. M. "Very-High-Level Parallel Programming in Crystal". Yale University Department of Computer Science Technical Report number YALEU/DCS/RR-506, Dec. 1986.

[8] Clark, K. and S. Gregory. "PARLOG: Parallel Programming in Logic". *ACM Transactions on Programming Languages and Systems*, Jan. 1986 (**8:1**). pp. 1–49.

[9] Clocksin, W. and C. Mellish. *Programming in Prolog*. Berlin: Springer-Verlag, 1981.

[10] Dewar, R. *et al.* "Programming by Refinement, as Exemplified by the SETL Representation Sublanguage". *ACM Transactions on Programming Languages and Systems*, July 1979 (**1:1**). pp. 27–49.

[11] Ellis, J. *BULLDOG: A Compiler for VLIW Architectures*. Ph. D. Dissertation, Department of Computer Science, Yale University, Feb. 1985.

[12] Even, S. *Graph Algorithms* (Chap. 5). Potomac, Md: Computer Science Press, 1979.

[13] Frenkel, K. "Evaluating Two Massively Parallel Machines". *Communications of the ACM*, Aug. 1986 (**29:8**). pp. 752–758.

[14] Gallier, J. and S. Raatz. "SLD-Resolution Methods for Horn Clauses with Equality Based on E-Unification". Proceedings of the 1986 Symposium on Logic Programming, Sept. 1986, IEEE Computer Society. pp. 168–179.

[15] Gelernter, D. "Domesticating Parallelism" (Guest Editor's Introduction). *Computer*, Aug. 1986 (**19:8**). pp. 12–16.

[16] Gosling, J. "Algebraic Constraints". Carnegie-Mellon University Department of Computer Science Technical Report number CMU-CS-83-132, May 1983.

[17] Hoddinott, P. and E. Elcock. "PROLOG: Subsumption of Equality Axioms by the Homogeneous Form". Proceedings of the 1986 Symposium on Logic Programming, Sept. 1986, IEEE Computer Society. pp. 115-126.

[18] Hoffman, C. and M. O'Donnell. "Programming with Equations". *ACM Transactions on Programming Languages and Systems*, Jan. 1982 (**4:1**). pp. 83-112.

[19] Josephson, A. and N. Dershowitz. "An Implementation of Narrowing: The RITE Way". Proceedings of the 1986 Symposium on Logic Programming, Sept. 1986, IEEE Computer Society. pp. 187-197.

[20] Kernighan, B. and S. Lin. "An Efficient Heuristic Procedure for Partitioning Graphs". *Bell System Technical Journal*, Feb. 1970 (**49:2**). pp. 291-307.

[21] Li, P. and A. Martin. "The Sync Model: A Parallel Execution Method for Logic Programming". Proceedings of the 1986 Symposium on Logic Programming, Sept. 1986, IEEE Computer Society. pp. 223-234.

[22] Nicolau, A. and J. Fisher. "Using an Oracle to Measure Potential Parallelism in Single Instruction Stream Programs". Proceedings of the ACM SIGMICRO 14th Microprogramming Workshop, Oct. 1981. pp. 171-182.

[23] Padua, D., D. Kuck, and D. Lawrie. "High-Speed Multiprocessors and Compilation Techniques". *IEEE Transactions on Computers*, Sept. 1980 (**C-29:9**). pp. 763-776.

[24] Seitz, C. "The Cosmic Cube". *Communications of the ACM*, Jan. 1985 (**28:1**). pp. 22-33.

[25] Shapiro, E. "Concurrent Prolog: A Progress Report". *Computer*, Aug. 1986 (**19:8**). pp. 44-59.

[26] Steele, G. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* Ph. D. Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Aug. 1980.

[27] Sutherland, I. "SKETCHPAD: A Man-Machine Graphical Communication System". Massachusetts Institute of Technology Lincoln Laboratory Technical Report number 296, Jan. 1963.

END

DATE

FILMED

MARCH

1988

DTIC